PARALLELIZATION OF LOCAL SEQUENCE ALIGNMENT (SMITH-WATERMAN ALGORITHM)

PROJECT REPORT

Submitted by

V V SAI DILEEP (18BCE0419)
G S S N ROHITH (18BCE0431)
S HARSHITH (18BE0419)

Guided by

PROF. DEEBAK B.D.

B. Tech

In

Computer Science and Engineering

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Vellore-632014, Tamil Nadu, India

School of Computer Science and Engineering

Table of Content

Problem Addressed:

Genome[a] is an emerging field, constantly presenting many new challenges to researchers in both biological and computational aspect of application. Sequence comparison is a very essential and important operation. They detect similar or identical parts between two sequences called the query sequence and the reference sequence.

Parallel and Distributed computing is the future of technology. All products and their fundamental concepts are being shifted to a parallel computing model. Everybody would agree that serial computing is easy to implement and use, but simply not efficient enough for industry-level purposes.

Due to this reason, day by day higher number of industries are providing and using cloud solutions which work on the basis on parallel and distributed computing.

To cope up with the fast-paced improvement in technology, one must also become familiarized with this domain, and hence this project.

Gene sequencing problem is one of the major issues for researchers regarding optimized system models that could help optimum processing and efficiency without introduction overheads in terms of memory and time. Bioinformatics and computational biology is a latest multidisciplinary field which explains many aspects of the fields of computer science, while computational biology harnesses computational approach and technologies to respond biological questions conveniently.

Prior Research:

The prior research involved in this paper includes research on Smit Water-Man Algorithm, parallel and distributed computing and the base paper. Researchers are compelled to use Smith-Waterman (SW) algorithm due to space and time constraints, thereby losing significant amount of sensitivity.

Parallelization is a possible solution, though, till date, the parallelization is restricted to database searching through database fragmentation.

In recent years, genome projects conducted on a variety of organisms generated massive amounts of sequence data for genes and proteins, which requires computational analysis. Sequence alignment shows the relations between genes or between proteins, leading to a better understanding of their homology and functionality. Sequence alignment can also reveal conserved domains and motifs.

For distantly related sequences, which remain undetected by BLAST with SW can be used. Again, for megabase-scale sequences, when SW becomes computationally intractable, the proposed method can still align them reasonably fast with high sensitivity.

Significance:

The Smith-Waterman algorithm looks for sequence subsets that best match in two large sequences. This algorithm will execute in O(mn) time where m is the length of the main sequence while n is the length of the match sequence. We fix the length of the main and match sequences to be the same, so the algorithm effectively runs in O(n 2) time.

The first step is to generate a similarity matrix. This similarity matrix decides if a pair of codons, a triplet of adjacent DNA nucleotides that code for proteins, are either exact matches, similar matches that are distinct but serve the same function, or dissimilar matches. This can be generalized as a function sim(a, b) that will return an exact match score, a similar score, or a no match score.

This algorithm works with a dynamic programming matrix A. This matrix is formed for each element Aij by comparing element i in the main sequence and j in the match sequence using the sim(a, b) function to derive a score that is added to element A(i−1)(j−1) of the score matrix representing the score if the current sequence were matched. A score is derived for a gap in the main sequence by subtracting a gap penalty from element Ai(j−1) in the score matrix.

A score for a gap in the match sequence is also derived by taking element A(i−1)j and subtracting the gap penalty. These scores, for the gap in main, the gap in match, and a similarity value between main and a match, are all compared and the highest score is the score used for element Aij in the matrix to a minimum score of 0.

Additionally, our implementation used the implementations which keeps a pair of additional arrays to map a changing gap penalty to penalize small gaps over large gaps. This is done using an additional array where the previous match against the main sequence is penalized with a gap start penalty, while the previous gap penalty, represented by E(i−1) is penalized with a gap extension value. Match has a similar array where previous gap penalties in the match sequence are located in F(j−1). These gap values are compared against the matching value rather than a fixed gap penalty. This is done on the observation that in nature a genome usually sees larger gaps rather than smaller gaps.

Introduction:

Genome[a] is an emerging field, constantly presenting many new challenges to researchers in both biological and computational aspect of application. Sequence comparison is a very essential and important operation. They detect similar or identical parts between two sequences called the query sequence and the reference sequence.

The global and local alignments are the most prevalent kinds of sequence alignment. In global alignment, we find the superior counterpart between parts of the sequences. On the other hand, local alignment algorithms try to match parts of sequences and not the entirety of them.

Local alignment is faster than global alignment, due to the lack of need to align the entire sequences. In our project, we would be implementing the Smith-Waterman Algorithm in a serial and parallel manner to for comparison and analysis. As common sense suggests, the parallel implementation should execute and provide the same result as the serial implementation but in a lesser amount of time.

Advances in genomics have triggered a revolution in discovery-based research and systems biology to facilitate understanding of even the most complex biological systems such as the brain.

Literature Review:

[1] Searching databases of DNA and protein sequences is one of the fundamental tasks in bioinformatics. The Smith-Waterman algorithm guarantees the maximal sensitivity for local sequence alignments, but it is slow. It should be further considered that biological databases are growing at a very fast exponential rate, which is greater than the rate of improvement of microprocessors. This trend results in longer time and/or more expensive hardware to manage the problem. Special purpose hardware implementations, as for instance super-computers or field programmable gate arrays (FPGAs) are certainly interesting options, but they tend to be very expensive and not suitable for many users.

[2] For the above reasons, many widespread solutions running on common microprocessors now use some heuristic approaches to reduce the computational cost of sequence alignment. Thus a reduced execution time is reached at the expense of sensitivity. FASTA (Pearson and Lipman, 1988) and BLAST (Altschul et al., 1997) are up to 40 times faster than the best known straight forward CPU implementation of Smith-Waterman.

[3] A number of efforts have also been made to obtain faster implementations of the Smith-Waterman algorithm on commodity hardware. Farrar exploits Intel SSE2, which is the multimedia extension of the CPU. Its implementation is up to 13 times faster than SSEARCH (a quasi-standard implementation of Smith-Waterman).

[4] To our knowledge, the only previous attempt to implement Smith-Waterman on a GPU was done by W. Liu et al. (2006). Their solution relies on OpenGL that has some intrinsic limits as it is based on the graphics pipeline. Thus, a conversion of the problem to the graphical domain is needed, as well as a reverse procedure to convert back the results. Although that approach is up to 5 times faster than SSEARCH, it is considerably slower than BLAST.

[5] There are many existing tools for sequence alignment. Among those, FASTA2 and BLAST3 are two commonly used ones, where the time complexity has been reduced through some heuristic algorithms. These heuristics algorithms obtain efficiency, however, at the expense of sensitivity. As a result, a distantly related sequence may not be found in a search using the above tools.

[6] Researchers have been worked on this issue through different approaches. For example, Fa Zhang, XiangZhen Qiao, and Zhi-Yong Liu presented a parallel SmithWaterman algorithm based on divide and conquer that can reduce running time and memory requirement. However, their method is also at the cost of losing sensitivity. Other methods that apply standard computer systems such as high performance supercomputers and computer clusters with software solutions for conducting the Smith-Waterman algorithm, although can achieve high sensitivity and reduce running time, are with extremely high cost. With the advance technology in the FPGA, a cost-efficient parallel implementation for the Smith-Waterman algorithm can be obtained

Methodology:

Smith-Waterman algorithm calculates the local alignment of two sequences. It guarantees to find out the best possible local alignment taking into account the specified scoring system. This includes a substitution matrix and a gap- scoring method . Scores consider match, mismatch and substitution. To measure the comparison between two sequences, a score be calculated as follows:

Given an alignment between sequences S0 and S1, the following values must be assigned, for each column:

- ma = (+5) [Match]
- mi = (-3) [Mismatch]
- G = (-4) [Gap]

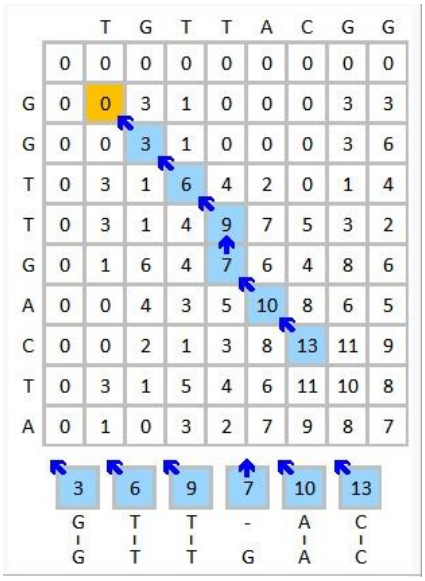
Procedure:

- Initialization of the matrix and consider two sequences A and B.
- Matrix filling with the suitable scores. The two sequences are set in a matrix form by means of A+1 columns and B+1 rows. The value in the first row and first column are set to zero.

$$M_{i,j} = Max \begin{cases} M_{i-1,j-1} + S_{i,j} \\ M_{i,j-1} + W \\ M_{i-1,j} + W \\ 0 \end{cases}$$

- The second and essential step of the algorithm is filling to entire matrix. To fill each and every cell it is important to know the diagonal values.
- Trace back the sequence for an appropriate alignment is trace backing; before that the maximum score obtained in the entire matrix has to be detected for the local alignment of the sequences.
    - It is likely to those maximum scores can be present in one or more than one cell, in such case there may be option of two or more alignments, and the best alignment can be obtained by scoring it.
- Tracing back begins from the position which has the highest value, pointing back with the pointers, consequently find out the possible predecessor, then go to next predecessor and continue until it reaches the score 0.
-

	-	C	G	T	G	A	A	T	T	C	A	G
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	5	1	5	1	0	0	0	0	0	0
A	0	0	1	2	1	10	6	2	0	0	5	1
C	0	5	1	0	0	6	7	3	0	5	1	2
T	0	1	2	6	2	2	3	12	8	4	2	6
T	0	0	0	7	3	0	0	8	17	13	9	7
A	0	0	0	3	4	8	5	4	13	12	18	14
C	0	5	1	0	0	4	5	2	9	18	14	15



The problem at hand was tackled with a modular approach. Eight functions were constructed, each of which would be explained as follows:

• **3.6.1 nElement** – This function is used to calculate the number of elements that have been found by the Smith Waterman Algorithm. Three conditions are given: One of which is to find out if the number of elements in the diagonal are increasing, decreasing or stable.

• **3.6.2calcFirstDiagElement** – This function is used to calculate the position of the maximum scored value in the matrix. This value needs to be found because the algorithm suggests that the backtracking to find the path should be started from this particular point.

• **3.6.3 similarityScore** – This function is used to find out the optimal order of execution based on three conditions, which are used to calculate the new values of left, upper and the diagonal elements. If the diagonal element > maximum element, Move diagonally upwards If upper element > maximum element, Move upwards If left element > maximum element, Move leftwards Every iteration, the values of maximum element is updated and inserting into the similarity and predecessor matrices.

• **3.6.4 matchMismatchScore** – This function is used to calculate a similarity function or the alphabet for a match or mismatch. If the value of the two elements are equal, it is a match, otherwise it's a mismatch.

• **3.6.5 Backtrack** – The purpose of this function is to modify the matrix that needs to be printed and helps us identify the path that needs to be taken to get the most optimum solution.

• **3.6.6 printMatrix** – It's a looped iteration implementation to display the matrix.

• **3.6.7 printPredecessorMatrix** - It is in this function in which we print the arrows depicting the path of local alignment.

• **3.6.8 Generate** – This function generates the two sequences A and B which would be locally aligned with each other. A random seed is used to ensure the reproducible nature of the output.

Contribution:

The authors of this report, i.e, Rohith, Harshith and Dileep, distributed work in terms of research, and shared the ideas presented in the various papers, materials and information about technologies available online. We are constantly supported by our teacher, Prof. Deebak B D, and he has helped us clear our concepts about parallel and distributed computing in computer systems. The report is also prepared with a joint effort by Dileep, Harshith and Rohith.

Code Implementation:

Serial.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include<time.h>

#define RESET   "\033[0m"
#define BOLDRED "\033[1m\033[31m"      /* Bold Red */

#define PATH -1
#define NONE 0
```

```
#define UP 1
#define LEFT 2
#define DIAGONAL 3

void similarityScore(long long int i, long long int j, int* H, int* P, long long i
nt* maxPos);
int matchMissmatchScore(long long int i, long long int j);
void backtrack(int* P, long long int maxPos);
void printMatrix(int* matrix);
void printPredecessorMatrix(int* matrix);
void generate(void);
long long int m = 11; //Columns - Size of string a
long long int n = 7;  //Lines - Size of string b

//Defines scores
int matchScore = 5;
int missmatchScore = -3;
int gapScore = -4;

char *a, *b;

int main(int argc, char* argv[]) {
    m = strtoll(argv[1], NULL, 10);
    n = strtoll(argv[2], NULL, 10);

    #ifdef DEBUG
    printf("\nMatrix[%lld][%lld]\n", n, m);
    #endif

    //Allocates a and b
    a = malloc(m * sizeof(char));
    b = malloc(n * sizeof(char));

    //Because now we have zeros
    m++;
    n++;

    //Allocates similarity matrix H
    int *H;
    H = calloc(m * n, sizeof(int));

    //Allocates predecessor matrix P
    int *P;
    P = calloc(m * n, sizeof(int));
    generate();
```

```
    //Start position for backtrack
    long long int maxPos = 0;

    //Calculates the similarity matrix
    long long int i, j;
    clock_t start, end;
    double cpu_time_used=0.0;
    start = clock();

    double initialTime = omp_get_wtime();

    for (i = 1; i < n; i++) { //Lines
        for (j = 1; j < m; j++) { //Columns
            similarityScore(i, j, H, P, &maxPos);
        }
    }

    backtrack(P, maxPos);

    #ifdef DEBUG
    printf("\nSimilarity Matrix:\n");
    printMatrix(H);

    printf("\nPredecessor Matrix:\n");
    printPredecessorMatrix(P);
    #endif

    //Gets final time
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    double finalTime = omp_get_wtime();
    printf("\nElapsed time: %f\n\n", finalTime - initialTime);
    // printf("\nElapsed time: %f\n\n", cpu_time_used);
    free(H);
    free(P);
    free(a);
    free(b);

    return 0;
}


void similarityScore(long long int i, long long int j, int* H, int* P, long long i
nt* maxPos) {

    int up, left, diag;
```

```
    //Stores index of element
    long long int index = m * i + j;

    //Get element above
    up = H[index - m] + gapScore;

    //Get element on the left
    left = H[index - 1] + gapScore;

    //Get element on the diagonal
    diag = H[index - m - 1] + matchMissmatchScore(i, j);


    int max = NONE;
    int pred = NONE;


    if (diag > max) { //same letter ↖
        max = diag;
        pred = DIAGONAL;
    }

    if (up > max) { //remove letter ↑
        max = up;
        pred = UP;
    }

    if (left > max) { //insert letter ←
        max = left;
        pred = LEFT;
    }
    //Inserts the value in the similarity and predecessor matrixes
    H[index] = max;
    P[index] = pred;

    //Updates maximum score to be used as seed on backtrack
    if (max > H[*maxPos]) {
        *maxPos = index;
    }

}


int matchMissmatchScore(long long int i, long long int j) {
    if (a[j-1] == b[i-1])
```

```
        return matchScore;
    else
        return missmatchScore;
}  /* End of matchMissmatchScore */


void backtrack(int* P, long long int maxPos) {
    //hold maxPos value
    long long int predPos;

    //backtrack from maxPos to startPos = 0
    do {
        if(P[maxPos] == DIAGONAL)
            predPos = maxPos - m - 1;
        else if(P[maxPos] == UP)
            predPos = maxPos - m;
        else if(P[maxPos] == LEFT)
            predPos = maxPos - 1;
        P[maxPos]*=PATH;
        maxPos = predPos;
    } while(P[maxPos] != NONE);
}


void printMatrix(int* matrix) {
    long long int i, j;
    for (i = 0; i < n; i++) { //Lines
        for (j = 0; j < m; j++) {
            printf("%d\t", matrix[m * i + j]);
        }
        printf("\n");
    }

}

void printPredecessorMatrix(int* matrix) {
    long long int i, j, index;
    for (i = 0; i < n; i++) { //Lines
        for (j = 0; j < m; j++) {
            index = m * i + j;
            if(matrix[index] < 0) {
                printf(BOLDRED);
                if (matrix[index] == -UP)
                    printf("↑ ");
                else if (matrix[index] == -LEFT)
                    printf("← ");
```

```
                else if (matrix[index] == -DIAGONAL)
                    printf("↖ ");
                else
                    printf("- ");
                printf(RESET);
            } else {
                if (matrix[index] == UP)
                    printf("↑ ");
                else if (matrix[index] == LEFT)
                    printf("← ");
                else if (matrix[index] == DIAGONAL)
                    printf("↖ ");
                else
                    printf("- ");
            }
        }
        printf("\n");
    }

}

 void generate(){
    //Generates the values of a
    long long int i;
    for(i=0;i<m;i++){
        int aux=rand()%4;
        if(aux==0)
            a[i]='A';
        else if(aux==2)
            a[i]='C';
        else if(aux==3)
            a[i]='G';
        else
            a[i]='T';
    }

    //Generates the values of b
    for(i=0;i<n;i++){
        int aux=rand()%4;
        if(aux==0)
            b[i]='A';
        else if(aux==2)
            b[i]='C';
        else if(aux==3)
            b[i]='G';
        else
```

```
            b[i]='T';
    }
}
```

Parallel.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include <time.h>
// DEFINING THE GLOBAL VARIABLES AND FIXED VARIABLES
#define RESET "\033[0m"
#define BOLDRED "\033[1m\033[31m"
#define PATH -1
#define NONE 0
#define UP 1
#define LEFT 2
#define DIAGONAL 3
#define min(x, y) (((x) < (y)) ? (x) : (y))
#define max(a,b) ((a) > (b) ? a : b)
// FUNCTION DEFINITIONS
void similarityScore(long long int i, long long int j, int* H, int* P, long long i
nt* maxPos);
int matchMissmatchScore(long long int i, long long int j);
void backtrack(int* P, long long int maxPos);
void printMatrix(int* matrix);
void printPredecessorMatrix(int* matrix);
void generate(void);
long long int nElement(long long int i);
void calcFirstDiagElement(long long int *i, long long int *si, long long int *sj);
// DEFINING THE MAIN VARIABLES
long long int m ; //Columns - Size of string a
long long int n ; //Lines - Size of string b
int matchScore = 5;
int missmatchScore = -3;
int gapScore = -4;
char *a, *b;
// MAIN FUNCTION FOR EXECUTION
int main(int argc, char* argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);
    m = strtoll(argv[2], NULL, 10);
    n = strtoll(argv[3], NULL, 10);
```

```
    #ifdef DEBUG
    printf("\nMatrix[%lld][%lld]\n", n, m);
    #endif
    a = malloc(m * sizeof(char));
    b = malloc(n * sizeof(char));
    m++;
    n++;
    int *H;
    H = calloc(m *n, sizeof(int));
    int *P;
    P = calloc(m * n, sizeof(int));
    generate();
    long long int maxPos = 0;
    long long int i, j;
    double initialTime = omp_get_wtime();
    long long int si, sj, ai, aj;
    long long int nDiag = m + n - 3;
    long long int nEle;
    #pragma omp parallel num_threads(thread_count) \
    default(none) shared(H, P, maxPos, nDiag) private(nEle, i, si, sj, ai, aj)
    {

        for (i = 1; i <= nDiag; ++i)
        {
            nEle = nElement(i);
            calcFirstDiagElement(&i, &si, &sj);
            #pragma omp for
            for (j = 1; j <= nEle; ++j)
            {
                ai = si - j + 1;
                aj = sj + j - 1;
                similarityScore(ai, aj, H, P, &maxPos);
            }
        }
    }
    backtrack(P, maxPos);
    #ifdef DEBUG
    printf("\nSimilarity Matrix:\n");
    printMatrix(H);
    printf("\nPredecessor Matrix:\n");
    printPredecessorMatrix(P);
    #endif
    double finalTime = omp_get_wtime();
    printf("\nElapsed time: %f\n\n", finalTime - initialTime);
    free(H);
    free(P);
```

```
    free(a);
    free(b);
    return 0;
}
long long int nElement(long long int i) {
    if (i < m && i < n) {
        return i;
    }
    else if (i < max(m, n)) {
        long int min = min(m, n);
        return min - 1;
    }
    else {
        long int min = min(m, n);
        return 2 * min - i + abs(m - n) - 2;
    }
}
// FUNCTION TO CALCULATE THE ELEMENT FROM WHICH THE ALIGNMENT MUST START
void calcFirstDiagElement(long long int *i, long long int *si, long long int *sj)
{
    if (*i < n) {
        *si = *i;
        *sj = 1;
    } else {
        *si = n - 1;
        *sj = *i - n + 2;
    }
}
// FUNCTION TO CALCULATE THE SIMILARITY SCORES OF THE TWO SEQUENCES
void similarityScore(long long int i, long long int j, int* H, int* P, long long i
nt* maxPos) {
    int up, left, diag;
    long long int index = m * i + j;
    up = H[index - m] + gapScore;
    left = H[index- 1] + gapScore;
    diag = H[index - m - 1] + matchMissmatchScore(i, j);
    int max = NONE;
    int pred = NONE;
    if (diag > max) { //same letter ↖
        max = diag;
        pred = DIAGONAL;
    }
    if (up > max) { //remove letter ↑
        max = up;
        pred = UP;
    }
```

```
    if (left > max) { //insert letter ←
        max = left;
        pred = LEFT;
    }
    H[index] = max;
    P[index] = pred;
    if (max > H[*maxPos]) {
        #pragma omp critical
        *maxPos = index;
    }
}
// FUNCTION TO CALCULATE THE MISMATCH SCORES FOR THE TWO SEQUENCES
int matchMissmatchScore(long long int i, long long int j) {
    if (a[j - 1] == b[i - 1])
        return matchScore;
    else
        return missmatchScore;
}

// THE MAIN BACKTRACKING FUNCTION FOR SEQUENCE ALIGNMENT
void backtrack(int* P, long long int maxPos) {
//hold maxPos value
    long long int predPos;
    do {
        if (P[maxPos] == DIAGONAL)
            predPos = maxPos - m - 1;
        else if (P[maxPos] == UP)
            predPos = maxPos - m;
        else if (P[maxPos] == LEFT)
            predPos = maxPos - 1;
        P[maxPos] *= PATH;
        maxPos = predPos;
    } while (P[maxPos] != NONE);
}

// FUNCTION TO PRINT THE MATRIX
void printMatrix(int* matrix) {
    long long int i, j;
    printf("-\t-\t");
    for (j = 0; j < m-1; j++) {
        printf("%c\t", a[j]);
    }
    printf("\n-\t");
    for (i = 0; i < n; i++) { //Lines
        for (j = 0; j < m; j++) {
```

```
            if (j==0 && i>0) printf("%c\t", b[i-1]);
            printf("%d\t", matrix[m * i + j]);
        }
        printf("\n");
    }
}
// FUNCTION TO PRINT THE MAIN ALIGNMENT MATRIX
void printPredecessorMatrix(int* matrix) {
    long long int i, j, index;
    printf(" ");
    for (j = 0; j < m-1; j++) {
        printf("%c ", a[j]);
    }
    printf("\n ");
    for (i = 0; i < n; i++) { //Lines
        for (j = 0; j < m; j++) {
            if (j==0 && i>0) printf("%c ", b[i-1]);
            index = m * i + j;
            if (matrix[index] < 0) {
                printf(BOLDRED);

                if (matrix[index] == -UP)
                    printf("↑ ");
                else if (matrix[index] == -LEFT)
                    printf("← ");
                else if (matrix[index] == -DIAGONAL)
                    printf("↖ ");
                else
                    printf("- ");
                printf(RESET);
            } else {
                if (matrix[index] == UP)
                    printf("↑ ");
                else if (matrix[index] == LEFT)
                    printf("← ");
                else if (matrix[index] == DIAGONAL)
                    printf("↖ ");
                else
                    printf("- ");
            }
    }
    printf("\n");
    }
}
// FUNCTION TO GENERATE THE SEQUENCES RANDOMLY
void generate() {
```

```
    srand(0);
    long long int i;
    for (i = 0; i < m; i++) {
        int aux = rand() % 4;
        if (aux == 0)
            a[i] = 'A';
        else if (aux == 2)
            a[i] = 'C';
        else if (aux == 3)
            a[i] = 'G';
        else
            a[i] = 'T';
    }
    for (i = 0; i < n; i++) {
        int aux = rand() % 4;
        if (aux == 0)
            b[i] = 'A';
        else if (aux == 2)
            b[i] = 'C';
        else if (aux == 3)
            b[i] = 'G';
        else
            b[i] = 'T';
    }
}
```

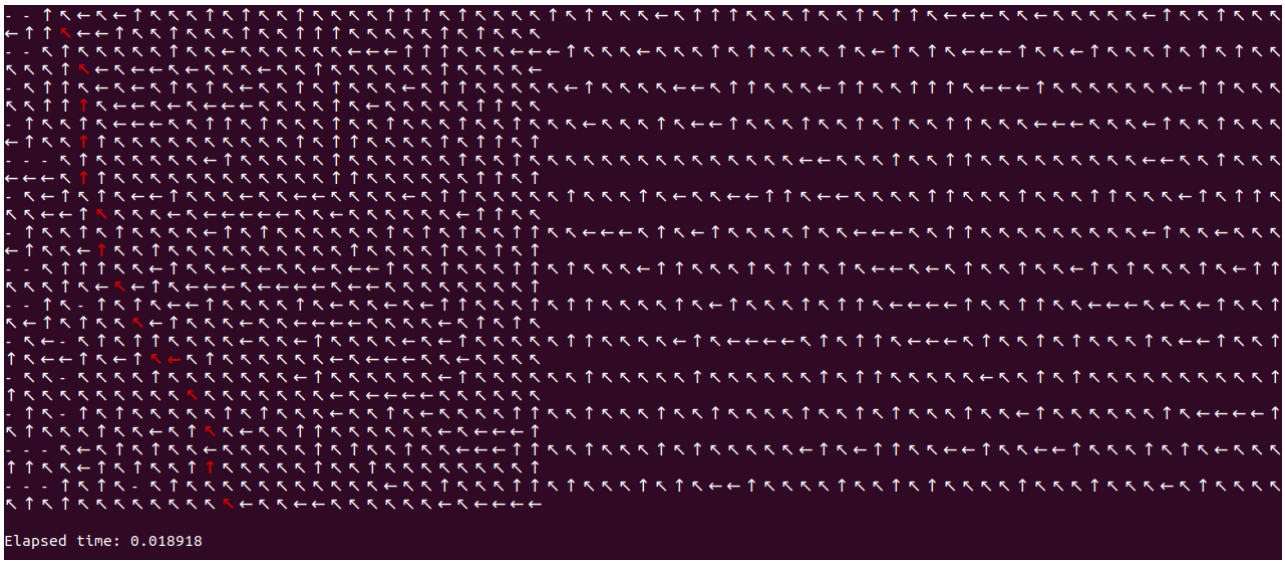
Result and Discussion:

For Serial and Parallel

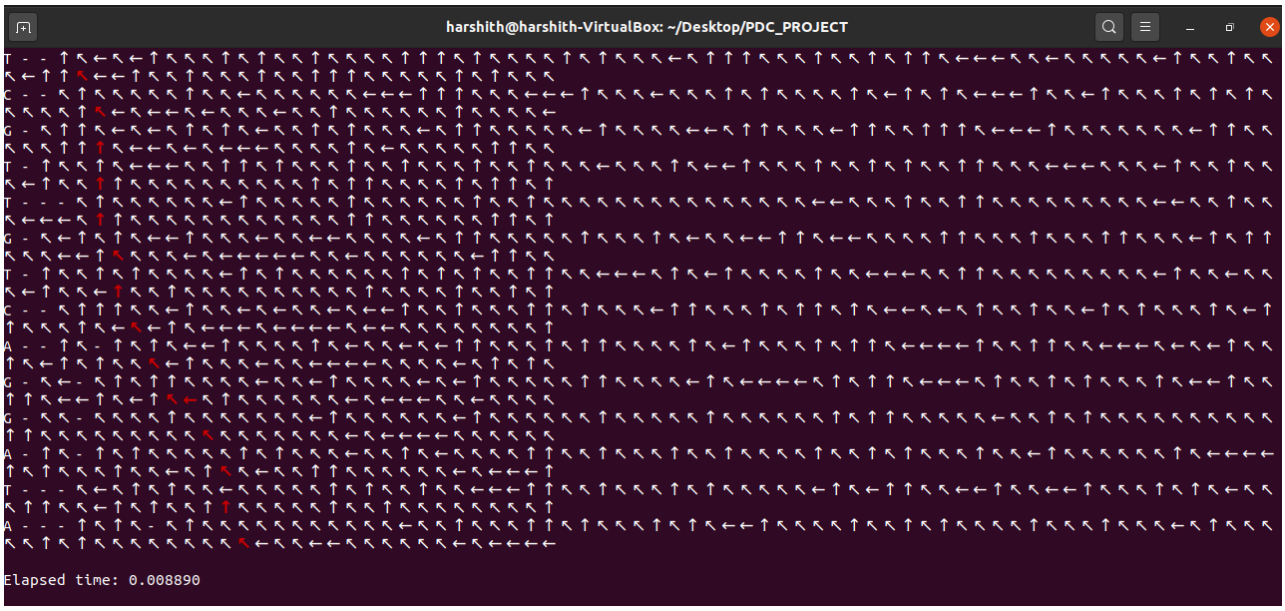
```
harshith@harshith-VirtualBox:~/Desktop/PDC_PROJECT$ ./serial 10 10

Matrix[10][10]

Similarity Matrix:
0       0       0       0       0       0       0       0       0       0       0
0       5       1       0       5       1       5       1       0       0       0
0       1       10      6       2       2       1       10      6       2       0
0       5       6       7       11      7       7       6       7       3       0
0       5       2       3       12      8       12      8       4       4       0
0       1       10      6       8       9       8       17      13      9       5
0       0       6       7       4       5       6       13      22      18      14
0       0       5       3       4       1       2       11      18      19      15
0       0       1       2       0       1       0       7       16      15      16
0       0       0       0       0       0       0       3       12      13      12
0       5       1       0       5       1       5       1       8       9       10

Predecessor Matrix:
- - - - - - - - - - -
- ↖ ← - ↖ ← ↖ ← - - -
- ↑ ↖ ← ← ↖ ↑ ↖ ← ← -
- ↖ ↑ ↖ ↖ ← ↖ ↑ ↖ ↖ -
- ↖ ↖ ↖ ↖ ↖ ↖ ← ← ↖ -
- ↑ ↖ ← ↑ ↖ ↑ ↖ ← ← ←
- - ↑ ↖ ↑ ↖ ↖ ↑ ↖ ← ←
- - ↖ ↖ ↖ ↖ ↖ ↖ ↑ ↖ ↖
- - ↑ ↖ - ↖ - ↑ ↖ ↖ ↖
- - - - - - - - ↑ ↖ ↖ ↖
- ↖ ← - ↖ ← ↖ ← ↑ ↖ ↖

Elapsed time: 0.002802
```

```
Elapsed time: 0.002802

harshith@harshith-VirtualBox:~/Desktop/PDC_PROJECT$ ./parallel 2 10 10

Matrix[10][10]

Similarity Matrix:
-       -       G       C       T       G       T       G       C       A       T       T
-       0       0       0       0       0       0       0       0       0       0       0
G       0       5       1       0       5       1       5       1       0       0       0
C       0       1       10      6       2       2       1       10      6       2       0
G       0       5       6       7       11      7       7       6       7       3       0
G       0       5       2       3       12      8       12      8       4       4       0
C       0       1       10      6       8       9       8       17      13      9       5
A       0       0       6       7       4       5       6       13      22      18      14
C       0       0       5       3       4       1       2       11      18      19      15
A       0       0       1       2       0       1       0       7       16      15      16
A       0       0       0       0       0       0       0       3       12      13      12
G       0       5       1       0       5       1       5       1       8       9       10

Predecessor Matrix:
 G C T G T G C A T T
 - - - - - - - - - -
G - ↖ ← - ↖ ← ↖ ← - - -
C - ↑ ↖ ← ← ↖ ↑ ↖ ← ← -
G - ↖ ↑ ↖ ↖ ← ↖ ↑ ↖ ↖ -
G - ↖ ↖ ↖ ↖ ↖ ↖ ← ← ↖ -
C - ↑ ↖ ← ↑ ↖ ↑ ↖ ← ← ←
A - - ↑ ↖ ↑ ↖ ↖ ↑ ↖ ← ←
C - - ↖ ↖ ↖ ↖ ↖ ↖ ↑ ↖ ↖
A - - ↑ ↖ - ↖ - ↑ ↖ ↖ ↖
A - - - - - - - ↑ ↖ ↖ ↖
G - ↖ ← - ↖ ← ↖ ← ↑ ↖ ↖

Elapsed time: 0.000433
```

Serial 100 x100



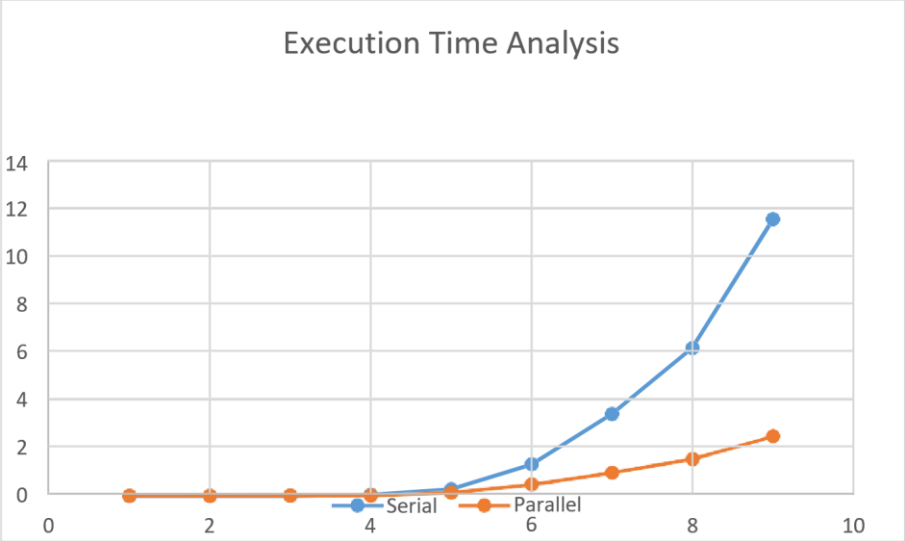
Parallel 100 x100



The code was run for various lengths of sequences, and the elapsed time was recorded in each case (as shown in the aforementioned outputs)

After tabulating all the execution times, we get:

Execution Times (seconds)	Serial	Parallel
10x10	0.001308	0.000767
20x20	0.00273	0.0019
50x50	0.0099	0.0086
100x100	0.0506	0.014
250x250	0.27707	0.128
500x500	1.324	0.474
750x750	3.44	0.9713
950x950	6.21	1.5444
1200x1200	11.62	2.498



Further Research:

As we can see in the above graph, for small lengths of the sequence, serial and parallel programs tend to give the output in almost the same amount of time. However, as the length increases, the execution time for serial implementation also increases exponentially, hence forming a steep graph.

However, the parallel implementation remains stable with not a high rise in the execution time because of the parallel execution of the task with two threads, making the process faster than its serial counterparts.

Base Paper and Reference Papers:

Base Paper:

- Ahmed Abdulhakim Al-Absi and Dae-Ki Kang. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Link: https://www.hindawi.com/journals/bmri/2015/807407/

Reference Paper:

[1] Cuong Cao Dang, Vincent Lefort, Vinh Sy Le, Quang Si Le, and Olivier Gascuel ,“Maximum likelihood estimation of amino acid replacement rate matrix”,

Bioinformatics. 2019, 27(19):2758-60.

[2] Frank Keul, Martin Hess , Michael Goesele and Kay Hamacher , “PFASUM: a substitution matrix from Pfam structural alignments” , June 5 2017

[3] Gary Benson ,Yozen Hernandez and Joshua Loving ,” A Bit-Parallel, General Integer-Scoring Sequence Alignment Algorithm” , 2004

[4] Vincent Ranwez and Yang Zhang ,” Two Simple and Efficient Algorithms to Compute the SP-Score Objective Function of a Multiple Sequence Alignment”, PLoS One, 2016 26

[5] Chao-Chin Wu, Jenn-Yang Ke, Heshan Lin, Wu-chun Feng, "Optimizing Dynamic Programming on Graphics Processing Units via Adaptive Thread-Level Parallelism", Parallel and Distributed Systems (ICPADS) 2011 IEEE 17th International Conference on, pp. 96-103, 2011.

[6] Chao-Chin Wu, Kai-Cheng Wei, Ting-Hong Lin, "Optimizing Dynamic Programming on Graphics Processing Units Via Data Reuse and Data Prefetch with Inter-Block Barrier
Synchronization", Parallel and Distributed Systems (ICPADS) 2012 IEEE 18th International Conference on, pp. 45-52, 2012.

[7] E. Rucci, C. García, G. Botella, A. De Giusti, M. Naiouf, M. Prieto-Matías, "State-of-the-Art in Smith-Waterman Protein Database Search on HPC Platforms" in Big Data Analytics in Genomics, Springer, pp. 197-223, 2016.

[8] Y. Liu, B. Schmidt, "Long read alignment based on maximal exact match seeds", Bioinformatic., vol. 28, pp. i318-i324, 2012.

[9] T. Rognes, E. Seeberg, "Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors", Bioinformatic., vol. 16, pp. 699-706, 2016.